Object Aware Execution Trace Exploration

Stefan Voigt, Johannes Bohnet, Jürgen Döllner
Hasso-Plattner-Institute – University of Potsdam, Germany
{stefan.voigt, bohnet, doellner}@hpi.uni-potsdam.de

Abstract

To understand software systems it is common practice to explore runtime information such as method calls. System behavior analysis can further be facilitated by additionally taking runtime data dependencies into account. In object oriented systems, a typical data dependency is the information about which objects are accessed by the traced method calls. To support software engineers in handling the massive amount of information that execution traces typically consist of, highly scalable visualizations are needed.

In this paper, we propose a trace-visualization technique that (a) explicitly visualizes both, method calls and object accesses, and (b) provides high scalability to handle large execution traces. With regard to the visualization technique proposed, we give a systematic overview of visual patterns that are to be expected and of their meanings with respect to system behavior. Additionally, we present the results of three case-studies to show how our approach facilitates developers in comprehending the behavior of complex C++ software systems.

1. Introduction

Adapting legacy software systems to changing requirements is often vital and time critical. Before developers can perform these maintenance tasks, they need to localize and understand the relevant parts of the system. These systems are typically large in terms of lines of code. Often previously performed changes have led to design anomalies, in many cases an up to date documentation is missing and each additional modification influences the future process of software maintenance. Thus, how well programmers understand software systems is *key to effective software maintenance and evolution* [14]. Understanding the rationale behind existing code is a challenge for many developers [12] and known to be a highly time-consuming task [6, 17]. As a consequence, the overall performance of software maintenance can be significantly improved by facilitating the process of program comprehension. During this process, developers can benefit from dynamic analysis which is able to provide accurate images of execution scenarios, because it can reveal the executed code and occurrences of late binding.

Many techniques of execution trace exploration only focus on the pure sequence of method calls, without considering the called objects. As calls can change object states and influence their future behavior, limiting the observation to the pure sequence of called methods is often not sufficient to understand object oriented systems. Dynamic analysis on the level of objects, i.e., by tracking and visualizing which objects are activated by method calls, aids developers in understanding nonlocal effects within software systems executions [13].

Due to the vast amount of data that execution traces typically consist of, highly scalable visualizations are needed to show the entities and their relations that are relevant for the developers at their individual tasks.

The main contribution of this paper is an exploration technique that enables developers to understand large execution traces, not only as sequences of method calls, but also as sequences of object activities. We attempt to achieve this goal by presenting an interactive and scalable visualization technique that is integrated into our visualization framework for analyzing complex C/C++ software systems [5].

The view provides high scalability and focuses on visualizing large parts of execution traces. It depicts information with respect to (a) the sequential order of method calls, (b) the sequential order of object activations and (c) the relationships between them. Multiple metrics can be visualized to customize the visualization to the developer's needs regarding their individual comprehension task. Although the view

978-1-4244-4828-9/09/$25.00 2009 IEEE Proc. ICSM 2009, Edmonton, Canada

focuses on depicting dynamic information, it additionally integrates information on the static structure of software systems.

Structure of the paper. Section 2 reports on related work. In Section 3 we briefly describe the data extraction mechanism and provide a model of the gathered information. Subsequently, in Section 4 we introduce our visualization technique. Section 5 reports on object aware exploration, including the results of three case-studies. We discuss the results in Section 6. Finally, we summarize our main contribution and outline future work in Section 7.

2. Related work

In the field of forward engineering, the Unified Modeling Language (UML) [3] is an established standard modeling notation. The language contains various diagram types that depict program behavior, such as the *sequence diagram* type and the *use case diagram* type. They are designed to model different aspects of software systems, but they are not that scalable to visualize large execution traces.

In the field of program optimization, Sherwood et al. use similarity matrices of basic blocks to visualize characteristics of large scale program behavior [16]. Focusing on program comprehension, a similar approach is used by Cornelissen and Moonen to visualize similarity patterns in method execution traces [7]. Both approaches focus on the detection and characterization of phases in the program execution.

Renieris and Reiss present a visualization technique that maps the sequence of call stacks onto a spiral view [15]. This visualization gives overview of the execution trace and can be combined with other views such as ones depicting call sequences in detail or visualizing source code.

Cornelissen et al. visualize execution traces using interactive sequence diagrams that focus on interactions at the class level [8]. They synchronize this sequential view with a circular view, which depicts the static structure of the software system within the surrounding border, while interactions between classes are depicted as border connecting lines within the circle. Among others, they focus on the detection of outliers in execution traces. To visualize the outliers while displaying a large number of method calls, they use techniques of antialiasing.

The approaches listed above [7, 8, 15, 16] – except UML – differ from our approach as they are not aware of objects.

Deelen et al. focus on the comprehension of class interactions [10] using a call graph and a timeline view at the class level. The timeline view shows the class interactions in a sequential way. As different points in time are selected in the timeline view, the call graph can encode different information on the classes such as the number of objects or the number of received calls. The nodes display several metrics such as the number of objects that exists at a given point in time. Program executions can be simulated by a token that moves along the edges to display how messages are exchanged between classes. As in contrast to our approach the view focuses on visualizing interactions at class level, it does not identify individual objects or methods.

De Pauw et al. create pattern based visualizations to allow programmers to explore program executions at varied levels of abstraction [9]. Their execution pattern view depicts objects and messages that are sent between them. In contrast to our work, they focus on reducing the massive amount of data by generalizing reoccurring execution patterns.

Greevy et al. visualize the dynamics of object oriented systems [11]. They introduce a 3D visualization that allows software developers to step through traces and thereby to inspect the current state of the program. The visualization encodes individual objects and permits developers to explore interactions at object level. However, it is limited in that it visualizes a single program state at one moment only.

Lienhard et al. discuss nonlocal effects in object oriented systems as a result of object aliasing and describe the consequences of these effects for program comprehension [13]. They analyze the object flow to detect runtime dependencies caused by activations and modifications of objects in different phases of execution. To visualize these dependencies, they introduce the object dependency graph.

In contrast to the approaches above, we focus on the exploration of the roles that individual objects have by displaying which method executions they relate to. Thereby, in difference to Lienhard et al. [13], we do not need to consider the different aliases the objects are referenced by.

3. Data Model and Data Extraction

In this section we present the data model that our analysis process is based on. Furthermore, we briefly introduce the mechanism we use to extract the required information.

3.1. Data Model

The data model (Figure 1) combines both static and dynamic information. The model describes program executions as sequences of *runtime events*. The model distinguishes between two types of events: *Method entries* and *method exits*. Events form pairs that relate to *method executions*. A method execution is always bound to a *method*. The term method is used as synonym for *class methods*, *member methods* and *functions* as they can occur in programming languages that mix procedural and object oriented paradigms such as C++. As a consequence, methods do not need to have a relationship to a class and their executions are not always bound to an object.

The model describes a relationship between method executions that is called *triggers*. Within each thread, this relationship forms a tree containing all method executions. As objects and methods relate to method executions, the relationship of triggering implies both relationships among objects and relationships among methods. Merging all nodes of the method execution tree that belong to the same method creates a directed graph describing a *calls* relationship among methods. In the same way, merging the nodes relating to the same object forms a *communicate to* relationship among objects. This relationship describes how messages are passed between objects.

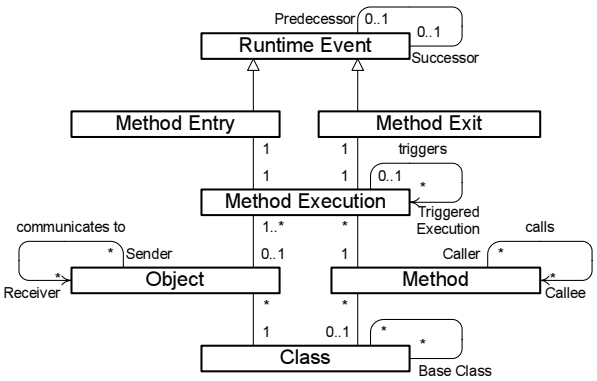

Figure 1. The data model.

3.2. Data Extraction

To extract the runtime information, we implemented a hybrid logging mechanism for C/C++ software systems, which combines both compiler based logging and binary patching. Compiler based logging means that the compiler inserts hook functions that implement logging functionality for each method entry and exit of the software system under study, without the need for source code modifications. Compiler based logging is provided by numerous compilers, e.g., gcc or Microsoft Visual Studio, therefore this approach is applicable to multiple platforms. The logging mechanism is combined with a binary patching mechanism that replaces call instructions leading to hook functions with no-operation instructions. This allows developers to choose which methods to log without the need for recompiling the program under study. This is an important prerequisite for the applicability of analysis tools based on the logging mechanism due to typically long recompilation times. The logging process results in a sequence of events per thread that either represent the control flow into or out of a method. Each logged event contains the following set of information:

- event type (entry, exit),
- thread of execution,
- method address,
- return address,
- this-pointer address (if being an object's method),
- timestamp.

In a post-processing step, the execution trace is combined with debug information to add static information to the model, i.e., information on classes and methods. Additionally, this-pointer addresses are used to extract object identities. Thereby, the extraction mechanism is aware of the twofold ambiguousness between this-pointer addresses and object identities. Due to multiple-inheritance, several valid this-pointer addresses can belong to the same object. Additionally, caused by both object containment and the limited lifecycle of objects, different objects can share (simultaneously or successively) this-pointer addresses.

Compared with our previous work [5], which is not aware of objects, additionally extracting this-pointer addresses results in reading one additional register and persisting 4 additional bytes for each logged method entry. This is only few overhead and does not recognizably slow down the system under study while extracting execution traces. The only drawback of logging the additional data is the increased complexity of the post-processing step.

4. Visualization Concept

In this section we introduce our visualization technique, the *event matrix*. In the following we describe how events are placed in the matrix.

Figure 2 a) and b) provide different perspectives on an execution scenario containing 7 method executions, i.e., 14 runtime events. Figure 2 a) visualizes method executions and triggering relationships, whereas b) is an UML [3] sequence diagram displaying object interactions. The matrix combines both perspectives on the execution scenario. Therefore, as shown in c), it is split along the diagonal and provides two sub views: the object sequence view in the lower left and the method sequence view in the upper right.

In the method sequence view, the horizontal alignment of events conforms to their sequential order and their vertical alignment describes their relationship to methods. In object sequence view, the vertical alignment describes their sequential order and the horizontal alignment describes their relationships to objects.

In contrast to Figure 2 a) and b), the spacing between methods and objects is not uniform. Instead, for each method and object, the first related event is placed at the diagonal. The subsequent related events are placed at the same row respectively at the same column. The rationale is that these methods and objects are grouped by their first activity. This means that objects and methods with similar initial events (with respect to their sequential number) are placed close to each other and therefore form clusters. This clustering supports developers in identifying execution phases. Additionally, the distance between events and the diagonal reflects the number of events passed since the first event that relates to the same object or method. This aids developers to identify objects and methods whose events span long ranges of the trace.

If more events shall be displayed than the resolution of the matrix allows, e.g., to get an overview of the whole trace, pixels of the matrix can span multiple events in both dimensions. Thereby, the effect of over-plotting occurs. We argue that while inspecting large parts of traces, the negative effects of over-plotting are not significant. The reason is that analyzing large parts of the trace mostly focuses on phase or pattern identification instead of identifying and understanding single events. Additionally, arranging the objects and methods according to their first event reduces the effect of over-plotting to affect only objects and methods whose first activities are close together.

Figure 2 d) displays a matrix visualizing the initialization phase of the Chromium browser [1]. Highlighting objects of the `View` class depicts that numerous objects are created at different phases. Most of the objects are long living and frequently accessed.

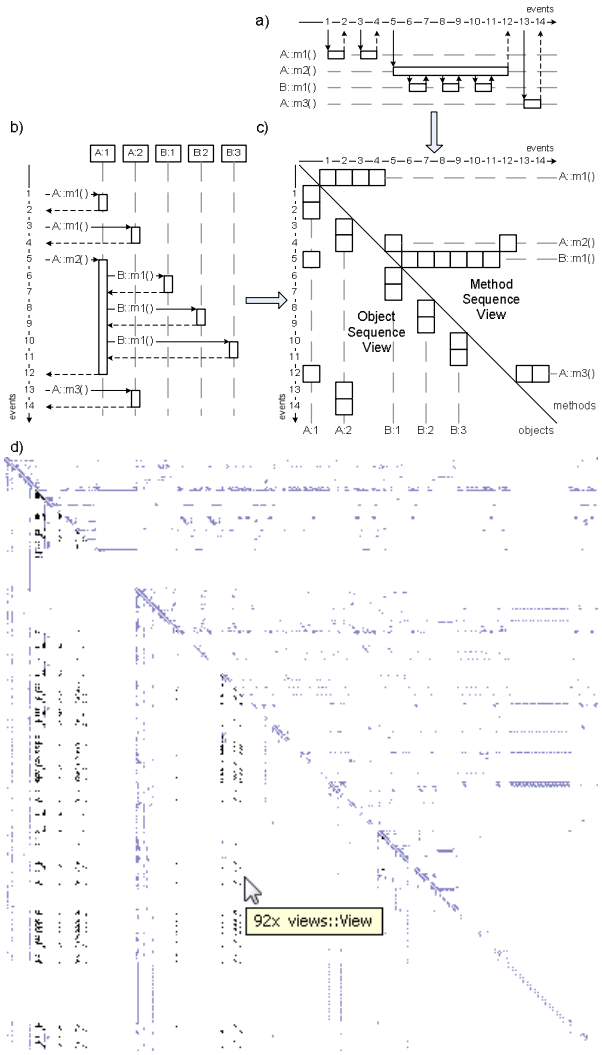

Figure 2. a) Sequence diagram diplaying method triggering relationships; b) UML sequence diagram displaying object communications; c) Schematic matrix visualization; d) Matrix visualization of a trace recorded from the Chromium software system.

Moving the mouse near an event results in a tool-tip providing additional information such as the object's or method's name.

Magnification. Parts of traces can be explored in more detail by means of zooming. This results in pixels spanning fewer events and consequently reduces over-plotting. Understanding of objects and methods that are active in the focused time range can be facilitated by contextual information, i.e., information describing their behavior in other parts of the trace.

We provide focus and context by creating a magnified area with low event density and a context area with high event density. In Figure 3 a) events are distributed uniformly. Consequently each cell of the grid spans the same number of events. In b), the cells are mapped to other areas of the matrix, but they contain the same events as their corresponding cells in a). This results in a lower event density in the gray focus area. Figure 3 c) and d) display a trace of the chromium browser's startup phase, containing 15944 events. The gray area depicted in c) (1) is magnified in d), that is, more detailed information on this part of the trace is provided. In c) 55 events (per dimension) are mapped to each pixel. Resulting from magnification, in d) the focus area contains 5 events per pixel.

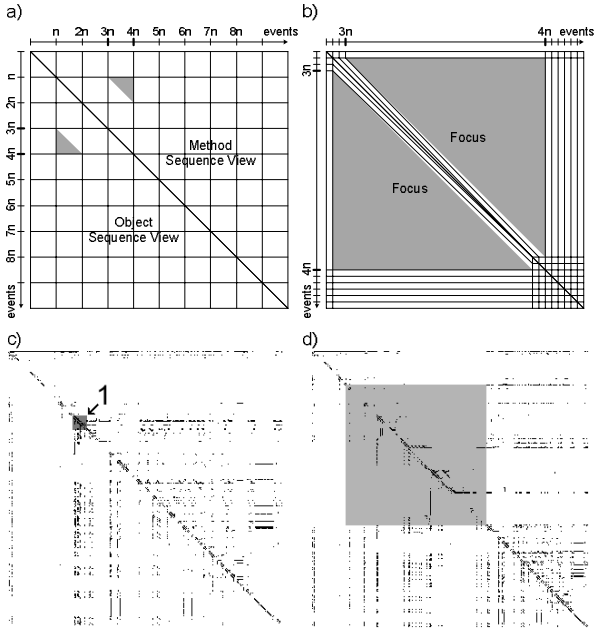

Figure 3: a) uniform event placement; b) magnification; c) chromium startup trace; d) chromium startup trace with magnification.

4.1. Trace Exploration

The visualization is integrated into our framework for visualizing complex software systems [5], which already contains different views that each focus on facilitating different aspects of the comprehension process. In the following, we describe how the matrix view can be involved in the trace-exploration process.

- The matrix view enables developers to recognize execution phases and large scale execution patterns. Thereby it supports them at identifying objects and methods whose activities are spread throughout various sections of the trace.
- The magnification allows a deeper inspection of single execution phases. Thereby the context information aids in identifying correlating phases, e.g., phases that contain similar events with regard to related objects or methods.
- While analyzing a small section of the trace within other views of our framework, the matrix view facilitates developers to understand the inspected section in a broader context. Therefore the section's time range is highlighted in the matrix view to visualize their position in the trace. Additionally, synchronization into the opposite direction enables the matrix view to be used as entry point for fine grained analysis.
- Different events can be highlighted in the matrix view with regard to different comprehension issues. As an example, highlighting events that relate to objects that are active in a certain section reveals data dependencies to other parts of the trace. Highlighting means fading the other events and disabling them for mouse interactions such as providing tool-tips to provide easier access to the objects of interest.
- Additional information about objects can be displayed such as the number of messages sent to the object, the number of objects belonging to the object's class or the threads objects are active in.
- Additional information about methods can be displayed such as their net and gross execution costs and their execution count.

The matrix addresses different comprehension issues developers may have by colorizing the events with respect to different aspects such as

- their type,
- the constness of the related method executions (i.e., is the execution changing the object's state),
- whether they are active in multiple threads,
- the system component they are part of.

To close the gap between method sequences and object sequences, moving the mouse on method events causes object events related to the method to be highlighted and vice versa. Additionally, highlighting is used to show the relationship among methods and among objects, e.g., by

- highlighting callers and callees,
- highlighting senders and receivers,
- highlighting executions that relate to base or child classes.

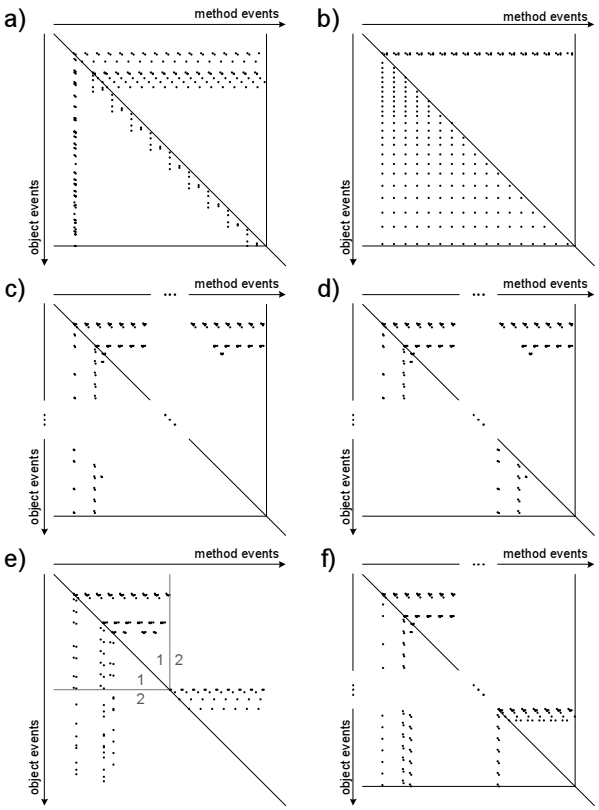

Figure 4: Different execution patterns; a) Highly active objects; b) Quadratic runtime complexity; c) Phases that are equal to each other in their executed functions and in their accessed objects; d) Phases that are equal to each other in their executed functions but differ in their accessed objects; e) Objects used in parallel; f) Phases with different functions that operate on the same objects.

4.2. Execution Patterns

In this section we describe different patterns that are expected to occur while explicitly visualizing object accesses. Additionally we explain how these patterns aid software developers to understand system behavior. For reasons of simplicity, the traces are only displayed schematically; usually they are much more complex. The patterns are labeled A – F and are visualized in Figure 4 a) – f).

A: The explicit visualization of object events reveals objects that are very active within the displayed phase. The pattern can occur at different granularity, i.e., within a short phase or spanning the whole trace. The identification of these objects facilitates trace understanding, as they either (a) contain key functionality and coordinate program behavior, (b) transfer information and state, or (c) provide (read-only) utility functionality and thus being a candidate to be filtered to de-clutter the trace.

B: The function view displays few functions being executed repetitively. With the information provided by the object view, it becomes visible that the objects are accessed with a quadratic runtime complexity.

C, D: In both c) and d) method events form a repetitive pattern. Additionally depicting the objects reveals that in d) the whole data the methods operate on has been exchanged. This can make a difference by means of considering the phases to be equal or not.

E: Figure 4 e) displays two phases (1, 2). Objects form pairs in Phase 1, i.e., they are called in similar patterns. In the matrix, this becomes visible as there are pairs of lines. In the second phase, one member of each pair is active. This pattern aids developers in understanding dependencies between objects (and their classes) such as parallel data structures. As half of the objects are only active in Phase 1, they might be intermediate and used to initialize objects of the other group. For a concrete maintenance task, knowledge about similar dependencies gives information on how to correctly extend the system, i.e., creating classes of both groups. The pattern is not restricted to pairs, i.e., larger groups of objects can correlate. For example, information of two data structures can be combined and put into a third one.

F: In Figure 4 f), two phases are displayed that seem to be independent from the method perspective, whereas the object view reveals that the phases are highly related to each other as they mainly operate on the same data. The knowledge of these dependencies facilitates programmers in avoiding to break seemingly unrelated code while performing maintenance activities.

5. Case Studies

We applied the visualization technique to traces of different software systems with regard to different comprehension questions on the execution scenario studied:

- What are the main execution phases?
- Which objects and methods are active within these phases?
- Which objects transfer state between phases?
- How do objects communicate with each other?

In the following, the results of the case studies are presented. Please note that all traces presented in the

case studies are filtered. They do not contain low level calls such as memory handling or calls to the standard template library. Reports on the size of traces always refer to the size after the filtering step.

5.1. Chromium Web Browser

Chromium [1] is a web-browser written in C/C++. It has been developed and maintained collaboratively by more than 100 developers and currently comprises approx. 1.2 MLOC. A typical usage scenario for the proposed visualization technique is to understand the basic principle of displaying a web-page. We argue that this principle can be understood by analyzing chromium's activities while rendering a small web page containing a bullet list with two list items.

We typed the URL into the address bar, started the logging mechanism, pressed the 'go' button, and stopped the logging mechanism after the page was rendered. By inspecting the function names, we identified one thread being responsible for rendering. This thread contains 48.558 runtime events and is displayed in Figure 5. We identified four main phases (A-D) that are to be described in the following.

Analyzing highly active objects (Pattern Areveals that there are two instances of `HTMLDocument` (1). One is active in Phase A; the other is active in Phases B-D. We conclude that in Phase A the browser's start page is rendered.

Throughout Phase B, an `HTMLParser` and a `Tokenizer` (2) are frequently called. They contain key functionality of this phase and coordinate the program's behavior (Pattern A). Within this phase, the second `HTMLDocument` and its complete tree-structure is created, among them

- 1 `HTMLHtmlElement` object
- 1 `HTMLBodyElement` object
- 1 `HTMLParagraphElement` object
- 1 `HTMLLuListElement` object
- 2 `HTMLLiListElement` objects (3)
- 2 `Text` objects

Thereby, for each of these `Node` elements, objects are created whose classes derive from `RenderObject`, such as `RenderBlock` or `RenderText`. This is a parallel data structure as described in Pattern E.

In Phase C, the previously created nodes and the render objects are active. We find that within this phase the page is loaded, i.e., the size and position of the `RenderListItems` are calculated. The method

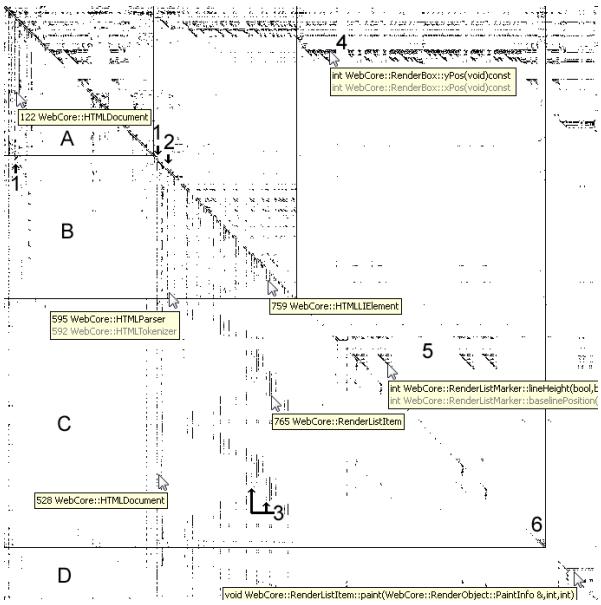

Figure 5. Chromium browser rendering an HTML page.

view of the matrix reveals that within Phase C, numerous methods of Phase A are executed (4), but additional methods are also executed (5). These methods are specific for calculating the size and position of `RenderListItems` not part of the start page being displayed before. The object view reveals that phase A and C share only few active objects, although they have high similarity in the executed methods (Pattern D). After the calculation of the position information, several objects are informed that the loading is finished (6).

In Phase D, the HTML page is finally rendered. In contrast to Phase C, only `RenderObjects` are active, but no objects that represent the document structure. The `HTMLDocument` itself constitutes an exception as it is massively called within this phase. Analyzing this exceptional behavior using the method view reveals that each `RenderObject` calls the method `documentElement` to check whether the node that they represent is the document's root.

Summary of the results. The explicit visualization of methods and objects allowed us not only to identify phases and to name the objects that are active within, but also to understand how objects interact with each other to create system functionality. Among others, we identified two parallel data structures: one contains the document itself as a result of the parsing process and the other one describes how the elements are displayed on screen. Additionally, we quickly grasped which objects are active in several phases and consequently understood how they transfer state among phases.

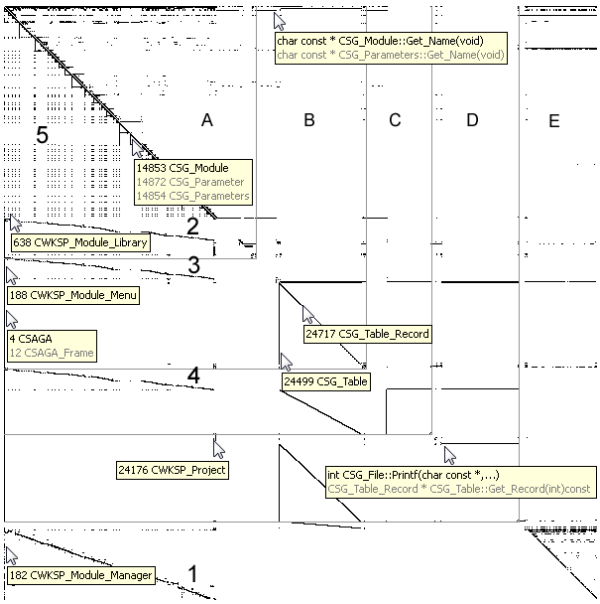

Figure 6. SAGA trace covering a complete application execution.

5.2. SAGA GIS

The System for Automated Geoscientific Analysis (SAGA) [2] is written in C/C++ and comprises more than 160KLOC. The application consists of several modules, among others to simulate fire spreading or to perform terrain analysis. Within this case study, we explore how modules are initialized and how they interact with the main application. We log a complete execution, i.e., (a) the startup, (b) selection and execution of a non-interactive simulation module, (c) displaying the simulation results and (d) committing to store the simulation results when (e) closing the application.

Figure 6 displays a matrix visualization of the execution (357.752 runtime events) that enables us to quickly find 5 phases (A-E) that relate to the activities mentioned above (a-e). The order of the last phases is switched as the results are stored before the application is shut down.

Within Phase A, a `CWSKP_Module_Manager` loads the module libraries, which themselves comprise several modules. On initialization, each module creates a list of parameters that configures its execution. The created module- and parameter-objects last until the application is shut down (1). They are traversed 3 times in a similar pattern, that is, when the module-menu is created (2), when it is used to start a module (3) and when it is used to show the calculation results (4). The reason is that the module menu is updated and the object-names are requested.

We identified the module libraries being accessed with quadratic runtime complexity (5), i.e., when a library is created, its name is compared with all existing libraries (Pattern B).

Phase B comprises the execution of the module, which results in the creation of a `CSG_Table` and its records. The `CSG_Table` is highly active (Pattern A). Although the module is non-interactive, the `CSAGA_Frame` is active within this phase. Looking at the method view reveals that a callback-mechanism is used to update the progress bar.

In Phase C, the content of the table is displayed. A `CVIEW_Table_Control` object is created and filled with the records created in the previous phase (Pattern F). Within this phase, the `CSG_Table` itself is not active.

Phase D comprises the requests to close the application, the commitment to save the table resulting from Phase B, and the process of saving. Within this phase, the `CWKSP_Project` identifies the table as being not yet saved, iterates the table's records, and calls a `CSG_File` to save them.

In the last phase, the objects are destroyed. The `CWSKP_Module_Manager` object controls the termination of the module libraries, which it created in the startup phase (Pattern A).

Summary of the results. Within this case study, we revealed which object creations and object interactions are triggered by user interactions. Additionally, we identified long-living objects whose state influences further system behavior, such as requesting the user to save data when closing the application. We identified repetitive phases that are equal to each other in both, their called methods and active objects. Additionally, as in the chromium case study, we identified object relationships, such as objects managing other objects.

5.3. Virtual Dub

Virtual Dub [4] is a video capturing and processing utility, which comprises more than 185KLOC of C/C++ code. Virtual Dub provides functionality of adding video filters that influence both the rendering output of Virtual Dub and the content of video files that can be saved to disk. For each frame the filters are processed consecutively. Within this case study we explore the handling of filters when rendering a video. The trace comprises adding four different filters (2:1 reduction, blur, flip horizontally, grayscale) to an already opened video and playing some seconds of the video.

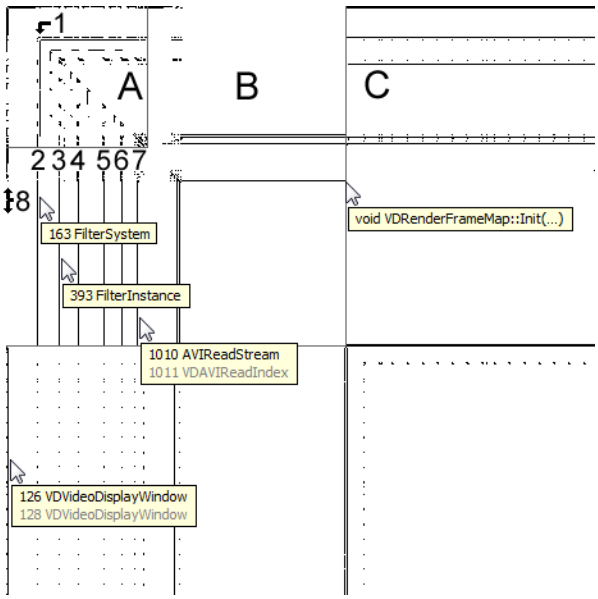

Figure 7. Virtual Dub trace.

Figure 7 displays the thread handling the graphical user interface (GUI), containing 3 phases (A-C) and 114.622 runtime events. Within Phase A, a `VDVideoFiltersDialog` (1) is highly active and has coordinating functionality (Pattern A). Four `FilterInstance` objects (3-6) are created, initialized and attached to a `FilterSystem` (2). In the end of the phase, an `AVIReadStream` (7) object and two `VDVideoDisplayWindow` (8) objects are active, that is, both video windows of Virtual Dub are updated as result of adding the filters.

The second phase is identified as initialization for playing the video. It differs from the Phase A in terms of the executed functions, but has high similarity in the active objects (Pattern F).

Highlighting events of triggering executions reveals that a large part of Phase B is triggered by a method execution that initializes a `VDRenderFrameMap`. Within this phase, the `FilterSystem` and the `FilterInstances` are highly active. The map sends the message `GetSourceFrame(int):int` repetitive to the `FilterSystem`. This causes the FilterSystem to send a message with the same signature to its filters, starting with the last inserted filter object. We conclude that when frames are rendered, filters can cause other frames being used as source. For each frame of the video, the map caches the source frame.

In Phase C, the video is played. Again, the filter system and the filters are recognized to be active. The method view reveals that only the source frames are requested, but no other messages are sent to the filters.

As the trace contains multiple threads, we checked whether the filter objects are active in other threads and found another thread comprising activities of the filter objects. Within this thread, which contains a repetitive pattern, the message `RunFilters` is sent to the filter system, which sends the message `Run` to the four filter objects. Further analysis turns out, that the message to the filter system originates from a `VDDubProcessThread` while executing the method `WriteVideoFrame`. We conclude that this thread creates the contents of the frames that are rendered by the GUI thread.

Summary of the results. This case study shows that the visualization of objects is not limited to support developers in understanding how objects transfer state between different execution phases. Furthermore, tracking object activities in multiple threads aids in understanding how objects transfer state among multiple threads.

6. Discussion

We applied the visualization to three different software systems to prove its ability to aid software developers in understanding the behavior of software system with focus on the behavior of objects. Within the case studies, we showed how the visualization allows software developers (a) to recognize different execution phases, (b) to analyze them by means of object activities and method executions and (c) to gain knowledge of nonlocal effects by identifying objects that transfer state between phases or between threads. Nevertheless, there are some limitations that we will discuss in the following.

Intuitivity: In order to use screen space efficiently, the views differ in their orientation. The drawback is that developers may require some training in order to use the visualization.

Nonlocal effects: The goal of the object aware trace exploration is to understand nonlocal effects. However, only nonlocal effects of a specific execution scenario are covered. Phases that seem to be independent might share active objects within other execution scenarios. Additionally, there exist other ways of transferring state between phases:

- base types and structs that have public fields,
- external libraries such as databases,
- files,
- other applications.

Multiple threads: Within the last case study, we identified objects that transfer state between two threads and visualized one of them using a matrix

view. As the matrix needs lots of screen space, there are scalability concerns with regard to the number of threads being explored in parallel.

The matrix places a fix number of events per pixel, but within the program execution, events are not distributed uniform in time. Consequently, a given time period of the program execution can consume different amount of space in the matrices. This makes it difficult to correlate events between matrices that display different threads. Placing events in a way that time is equally distributed within both matrices will result in lots of unused space, if the respective threads differ in their start time.

7. Conclusion

Dynamic analysis such as the exploration of execution traces has the potential to give insights into internal processes of complex software systems. In this paper we present a visualization technique that focuses on facilitating developers to overcome two main challenges of understanding execution traces, namely

- to handle the massive amount of data dynamic analysis typically results in and
- to understand nonlocal effects in object oriented software systems.

We described several patterns to be expected and their meaning for understanding program executions.

Future work. We plan to extend our approach in different ways. Among others, we plan to add filter mechanisms that allow developers to analyze the lifecycles of objects under study to understand their roles. Additionally, comparing lifecycles of multiple objects of a class could draw an image of the typical behavior or reveal outliers. Another field is to extend scalability by means of supporting the exploration of multiple threads and their interactions.

8. References

[1] Chromium Open Source Project: http://dev.chromium.org.

[2] System for Automated Geoscientific Analysis: http://www.saga-gis.org.

[3] Unified Modeling Language: http://www.uml.org.

[4] Virtual Dub: http://www.virtualdub.org.

[5] J. Bohnet, S. Voigt, and, J. Döllner, „Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-Focused Views on Execution Traces “, In *Proceedings of the 16th International Conference on Program Comprehension*, IEEE, 2008, pp. 268-271.

[6] T.A. Corbi, “Program understanding: Challenge for the 1990s”, *IBM Systems Journal*, 28(2), 1989, pp. 294-306.

[7] B. Cornelissen, and L. Moonen, “Visualizing Similarities in Execution Traces”, In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis*, 2007, pp. 6-10.

[8] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. van Wijk, „Execution Trace Analysis through Massive Sequence and Circular Bundle Views” In *Journal of Systems & Software (JSS)*, Elsevier, 81(12), 2008, pp. 2252-2268.

[9] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman, “Execution Patterns in Object-Oriented Visualization”, In *Proceedings of the Conference on Object-Oriented Technologies and Systems*, USENIX, 1998, pp. 219 - 234.

[10] P. Deelen, F. van Ham, C. Huizing, and H. van de Watering, “Visualization of Dynamic Program Aspects”, In *Proceedings of the 4th International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, June 2007, pp. 39-46.

[11] O. Greevy, M. Lanza, and C. Wysseier, “Visualizing live software systems in 3d”, *In Proceedings of the Symposium on Software Visualization*, ACM, 2006, pp. 47-56.

[12] T.D. LaToza, G. Venolia, and R. DeLine, “Maintaining Mental Models: A Study of Developer Work Habits”, In *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, pp. 492-501.

[13] A. Lienhard, O. Greevy, and O. Nierstrasz, “Tracking Objects to Detect Feature Dependencies”, *In Proceedings of the 15th International Conference on Program Comprehension*, IEEE, 2007, pp. 59-68.

[14] A. von Mayrhauser and A.M. Vans. “Program Comprehension During Software Maintenance and Evolution”, *IEEE Computer*, 28(8), 1995, pp. 44-55.

[15] M. Renieris and S.P. Reiss, “Almost: exploring program traces”, In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation*, ACM, 1999, pp. 70-77.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior”, In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 45-57.

[17] N. Wilde, “Faster reuse and maintenance using software reconnaissance”, 1994. Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL.